Benchmarking Web Server Architectures: A Simulation Study on Micro Performance

Haiyong Xie, Laxmi Bhuyan, and Yeim-Kuan Chang
Department of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
yong@cs.ucr.edu

Abstract

As Internet expands, the number of application servers, especially Web servers, has been increasing exponentially. To improve the performance of Web servers, researchers have paid attention to and studied the Web server's macro-performance, namely, the response time and throughput, which can be perceived by end users directly. In this paper, we have produced a micro benchmark, ServBench, by studying the micro performance of the most widely used Apache Web server. The bottleneck functions are identified by profiling the Apache server running with a realistic workload. We select some of these functions as micro-benchmark programs and study their characteristics. We port the microbenchmark to SimpleScalar simulation environment. We obtain execution time, branch prediction and cache miss results for the microbenchmark as a function of various architectural parameters.

1. Introduction

Recent years have seen an explosive growth of the Internet. Web applications and Web servers are critical to the success of the Internet. To improve the performance of Web servers, researchers have studied the Web server's macro-performance, namely, the response time and throughput, which can be perceived by end users directly. These studies have led to many benchmarks such as SPECweb [24], WebStone [26], NetPerf [20], and WebBench[27]. However, most of the macro-performance bottlenecks such as protocol stack overhead and process management overhead actually stems from the operating systems. Other studies show that web servers spend about 85% of the cycles in executing operating system codes compared to only 9% by SPEC95 suite [25]. Hu et al [14] found that Apache spends only 20-25% of the total CPU time on user code. This means Apache spends most of the CPU time in the kernel of operating system.

A number of performance evaluation studies on web servers have been reported in the literature. Most of these studies characterize external performance of web servers, namely, how the web server interacts with the outside world, which is called macro-performance in this paper. The workloads either consist of mainly static web page accesses or many static web page access blended with a small percentage of CGI scripts that perform very simple computation functions. A number of performance evaluation methodologies have been suggested in the literature [11,13,16,17].

The studies on improving Web server's macro-performance focus on improvement of either the interactivity between the Web servers, the underlying operating systems, or disk I/O and network I/O. The studies in this field generally fall into three categories: operating system enhancement [2,6,7,8], server program improvement [1,5], and caching techniques [12,15].

A very limited number of studies focus on architectural performance of Web servers, which we call micro performance in this paper. Radhakrishnan and John [23] evaluated the performance of Apache Web server in terms of micro-architecture using hardware performance monitoring counters. They studied such architectural performance as CPI and cache miss rates for both static and dynamic workloads. Iyer studied the cache performance of single and dual-processor servers running SPECWeb99 benchmark by feeding traces through simulation models of CacheFlowII [16]. The micro performance is very important for us to fully understand the impact of micro-architecture on the Web servers.

Another motivation, which is more important, is that we believe macro-performance improvement has its physical limitations like input/output processing and that we cannot exceed these limitations. Flash Web server [21] is claimed to be the fastest Web server and it outperforms existing Web servers by up to 50%. There is still much room to improve the Web server's micro performance

which can lead to better macro-performance. In order to understand how to improve the micro performance, we need to explicitly study the behavior of frequently used functions that contribute greatly to the execution time. A far out research will then be to develop assembly language instructions (like Intel MMX) for these functions or to build specialized hardware units on the CPU for fast execution of these functions.

To better understand the micro performance of web servers, we build an experimental environment for measuring the internal performance of a common modern Web server, Apache [2]. We take advantage of gprof [11], which is a program of GNU suite Unix tools [12], to get detailed profiling information of the Apache server. Through profiling the web server program, we are able to evaluate the performance of the web server in terms of its function calls. We identify the most time-consuming functions and the most frequently called functions. These functions are the bottlenecks to the server's micro performance and comprise the kernel of the web server.

Having known how much time these functions spend and how frequently they are called, we extract the top 8 function calls from the Apache server program and use them as the micro benchmark, ServBench. To make these programs run in a real system, we also extract their corresponding data structures together with the functions. All the benchmark programs need workload to operate on. We add some workload builder functions to the server program. When Apache serves incoming requests, the workload builder automatically generates the workload for benchmark programs based on the actual processing of the requests.

To know better the characteristics of the benchmark programs, we port and run the programs in the simulator, SimpleScalar [8]. As far as we know, ours is the first attempt to port a Web server benchmark program to an execution-driven simulator. By means of simulation, we obtain the characteristics such as instruction level parallelism, instruction frequencies, and cache performance for the micro-benchmark as a function of various architecture parameters. These characteristics are of great help to the design of high performance Web servers and Web-server-specific network processors.

We find that the average code size of ServBench is an order of magnitude smaller than that of SPECint. Both have similar instruction set characteristics. However, ServBench has smaller basic-block sizes and nearly half of the branches are taken and half not taken. This fact makes better branch predication mechanism and lower miss rate very important to the performance. We find that the Apache Web server can benefit tremendously from instruction level parallelism (ILP) because of the inherent parallelism of the ServBench programs. Also, L1 instruction cache plays a more important role than data cache in increasing the number of instructions executed per cycle (IPC). IPC is not sensitive to the set-associativity of instruction cache. We are able to achieve higher IPC by using asymmetric L1 cache, by enlarging the length of instruction fetch queue and by adding more ALUs. However, 4 ALUs and an instruction fetch queue of length 8 are enough to enhance the micro performance.

The rest of this paper is structured as follows. Section 2 describes profiling information of the measurement and how we achieve the ServBench. This section also gives a brief description of the experimental setup, how the web server services the requests and how we measure the architectural and functional performance using httperf [18] and GNU profile tools [12]. Section 3 presents the ServBench based on the profiling data in Section 2. Section 4 presents characteristics of benchmark programs including the instruction level parallelism, instruction frequencies, and cache performance. Section 5 concludes our work and suggests some future work.

2. Micro performance Measurement and Profiling

2.1 Experimental Setup

To measure and profile Apache Web server in the level of function calls, we have established an experimental environment which is comprised of a Linux server running Apache Web server, and several clients running the Web server benchmarking tool, httperf. The server and clients are connected to each other by a dedicated Ethernet using a 100Mbps Ethernet switch. This ensures that both the server and clients have access to enough network bandwidth available thus both have feasible high throughput and low response time. We have carefully chosen the benchmarking parameters for httperf including the request rate, number of requests, and number of connections so that the server reaches its possible highest throughput with the lowest load.

To get detailed profiling information, the Apache Web server is compiled by gcc 2.91 with function profiling option and optimization level O2. We use O2 level optimization for the reason that the compiler only performs target-processor independent optimizations and does not exploit particular architectural features such as loop unrolling for superscalar architectures.

The simulated processor architecture in SimpleScalar tool set is a close derivative of MIPS architecture. In all the simulations, the default issue width is 4; the default L1 caches are 4-way set-associative 16KB separate instruction cache and data cache; the line size of L1 cache is 32 bytes; the L2 cache is a 4-way set-associative 512KB unified cache with line size of 64 bytes; and the simulator is configured as out-of-order execution.

In addition to setting up the experimental network, server and clients, we also build the realistic workload for

the server according to SPECweb99 [24]. The workload consists of static files of four classes as shown in Table 1.

Classes	File Sizes	Target Mix
Class 0	less than 1K	35%
Class 1	less than 10K	50%
Class 2	less than 100K	14%
Class 3	less than 1000K	1%

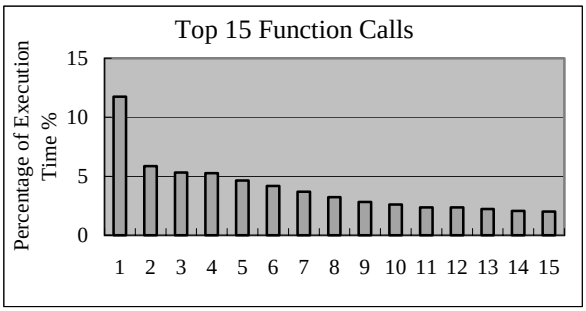
Table 1. File size mix of workload

We did not measure the performance with blended workloads which consist of both static and dynamic requests for a simple reason: our goal is to characterize the micro performance of the underlying processor and the internal performance of the server program. To study the micro and internal performance, we only need some simple but typical workloads which can be used to make the blended workloads. Dynamic requests always lead to the execution of some external programs such as Java or Perl CGI programs other than the Web server. The micro performance of those programs executed dynamically is not what we focus on.

2.2 Profiling Results

The clients in the experimental environment run httperf simultaneously to request a particular file class from the server. Apache Web server compiled with profiling options services the requests and writes the function profiling information to a specific binary output file. Later, the binary file can be converted to plain text file containing detailed profiling information using gprof. We collect all the function profiling information from the converted text file. We only pay attention to those functions that involve no disk I/O or network I/O activities directly since we focus on profiling the micro performance of Apache Web server.

After having collected profiling information for all the non-I/O functions, we rank the functions according to the percentage of execution time they account for. The top 15 most time-consuming functions account for almost 60% of execution time when the server services the HTTP requests, as shown in Figure 1.



	Function Name	%		Function Name	%
1	format_converter	12	9	check_hostalias	3
2	pstrdup	6	10	getword_white	3
3	run_method	5	11	process_item	2
4	config_log_transaction	5	12	conv_10	2
5	pstrcat	5	13	no2slash	2
6	palloc	4	14	getparents	2
7	table_get	4	15	get_module_config	2
8	invoke_handler	3			

Figure 1. Top 15 function calls

To our surprise, most of these functions are string processing related functions. Only a small part of these functions deal with the HTTP requests directly. This is feasible because HTTP protocol is a text-based protocol and the processing of HTTP protocol is essentially processing of strings. Some of these functions are sub-functions of others, for instance, process_item and conv_10 are sub-functions of config_log_transaction and format_converter, respectively.

3. ServBench

3.1 Selection of Benchmark Programs

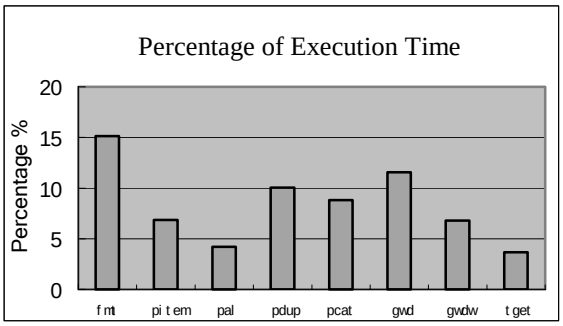
Most of the functions in Figure 1 have supporting sub-functions, for example, format_converter always calls conv_10 to convert integer numbers and conv_fp to convert floating-point numbers. Thus, we create fmt by combining the main function, format_converter, with such supporting functions as conv_10.

Some of the top 15 functions are just interfaces to a group of functions, for example, run_method and invoke_handler are called to invoke other functions indirectly using function pointers, which can not be identified by gprof. We do not consider these functions because they spend very little time in invoking other functions.

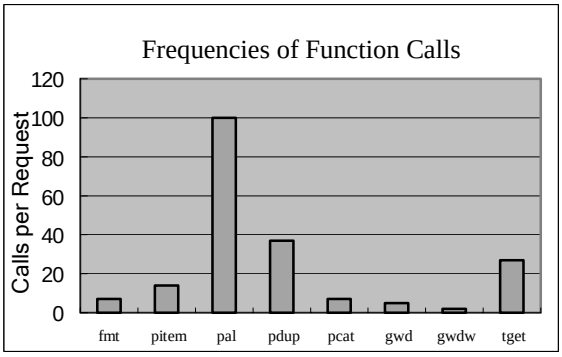
After having combined sub-functions with main functions and deleted interface functions such as run_method, we obtain eight sets of programs that are most time-consuming, fmt, pitem, pal, pdup, pcat, gwd, gwdw, and tget, as shown in Figure 2.1. These programs are ranked according to the percentage of execution time they spend. They account for 40% of total execution time (without considering disk and network I/O time). In this paper, we take these sets of programs as single functions for clarity.

These eight programs have different frequencies as shown in Figure 2.2. String allocation (pal) and duplication (pdup) functions are called 100 and 36 times respectively during Apache processes an incoming request. However, other functions such as fmt account for more execution time compared to pal although they are called less frequently. The reason is that fmt is about an order of

magnitude larger than pal and pdup in the size of source code and runtime kernel, as we will see in Section 5.2.



(1) Percentage of execution time



(2) Call frequencies

Figure 2. Benchmark programs: percentage of execution time and call frequencies

We build the micro-benchmark, ServBench, based on the above eight programs. There are several reasons for which we choose them as the benchmark programs. First, They are the most time-consuming elements of Apache server. If we want to improve the micro performance of Apache server, we will have to improve the performance of these functions because they are the bottleneck functions. Secondly, they are very frequently called when Apache processes the requests. Thirdly, benchmark programs should represent a wider application class in the domain of interest. The above functions are general functions to process the HTTP requests. Since HTTP protocol is basically a text-based protocol, all the request lines and header lines in the protocol payload are texts; to generate the response headers and log the requests are text-based as well. We believe every implementation of Web servers needs to process the text-based request lines and header lines very frequently. These string processing functions are representatives of HTTP protocol processing applications.

These programs can be divided into three groups. Group 1 has fmt and pitem as string format conversion programs; group 2 has pal, pdup, and pcat as string generation programs; group 3 consists of gwd, gwdw, and tget as string comparison programs.

3.2 String Format Conversion

String format conversion functions represent the operations of converting a number of values to a string. These functions are called to generate a response header or to log the corresponding request. String format conversion functions include fmt and pitem. Fmt is used to convert all other types of data to a string, e.g., converting an integer to its corresponding printable string, converting the request time to a string, and generating weak Etags for response headers. Pitem is used to generate the log entries for each of the incoming requests, for example, request line, request time, client address, and status of responses, etc.

3.3 String Generation

String generation functions, which include allocating a memory block for a new string, duplicating a string, and concatenating a number of strings, are very frequently called during the process of incoming HTTP requests. As mentioned before, HTTP protocol payload is text-based strings. To manipulate the payload, the content of payload which is comprised of many strings, has to be duplicated and stored in user space buffers. For instance, Web servers have to keep the state of a request in memory which may consist of the request string and some of the header strings. Web servers also need to log the requests which requires keeping some of the request strings in memory. These functions are also used in generating response headers.

String generation functions include pal, a string allocation function, pdup, a string duplication function, and pcat, a string concatenation function.

3.4 String Comparison

String comparison functions are very often called as well. These functions are used to extract a part from a long string which consists of many sub-strings separated by delimit characters such as blank space or colon. Tget, gwd, and gwdw are the three programs in this category. Gwd and gwdw are called to extract a “word” from a string which comprises many words separated by blank spaces or other predefined delimit characters. Tget is used to retrieve the corresponding value of headers from the HTTP requests. Since each request may have many headers, which again consists of may key strings and their corresponding value strings, tget is called to get the value string for a specific key string.

4. Benchmark Characteristics

We have selected the following general areas of characterization for further consideration: program code

sizes and kernel sizes, instruction set characteristics, instruction level parallelism, and cache performance.

4.1 Methodology

We use SimpleScalar to study the characteristics of these benchmark programs. SimpleScalar is an execution-driven simulator package commonly used by computer architecture researchers. SimpleScalar has its own C compiler (a modified version of GNU GCC) with associated utilities. We run the programs in the simulator and get the detailed performance data by changing the architectural parameters such as cache size, number of ALUs, etc.

All the programs have to have some data to deal with. To generate datasets for them, we insert some small functions into Apache server to gather all the data needed by each benchmark program. By doing so we are able to get the data that is dealt with by Apache derived from the realistic workloads, which is meaningful to characterizing the benchmark. This method has a potential valuable property: we are able to update the associated data for each benchmark programs very easily as the workload changes.

4.2 Program Kernel Size

Knowing the sizes of program kernels is useful for us to learn the static properties such as the number of lines of C code and size of compiled code, and dynamic properties such as instructions executed at least once and instructions accounting for 99% of execution time. We compare ServBench programs to SPECint programs as well.

Table 2 shows the size of the C source code and compiled executable of each benchmark program in both ServBench and SPECint. The object code size does not include dynamically linked libraries.

The average code size of ServBench programs is 28,679 bytes which is nearly an order of magnitude smaller than that of SPECint programs. The differences in code sizes of ServBench and SPECint programs come from the different environments where the applications or functions have been implemented and executed. String generation functions such as pal, pdup, and pcat are the most frequently referenced functions in Apache; they have rather simple functionalities compared to other function calls and applications. Other programs such as string format conversion and comparison functions are larger in code sizes in average. However, the SPECint programs are actual applications in real systems. They all have much more complex functionalities thus have much larger object code sizes.

ServBench	Code Size (C Lines)	Code Size		SPECint	Code Size (C Lines)	Code Size
Fmt	3,206	76732		126.gcc	206,000	1950000
Pitem	1396	20040		130.li	7,600	139000
Pal	326	9644		099.go	29,200	558000
Pdup	320	9628		134.perl	26,900	544000
Pcat	327	9992		124.m88ksim	19,900	404000
Gwd	427	75876		147.vortex	67,200	1150000
Gwdw	424	19560		132.ijpeg	31,200	594000
Tget	174	7960		129.compress	19,300	81700
Average	825	28679		Average	48700	678000

Table 2. Code sizes of ServBench and SPECint

ServBench	Instructions at Least once	Instructions For 99%		SPECint	Instructions at Least once	Instructions For 99%
Fmt	18400	1112		126.gcc	124246	15899
Pitem	1767	206		130.li	7341	408
Pal	660	198		099.go	12627	949
Pdup	657	226		134.perl	12313	875
Pcat	713	269		124.m88ksim	12284	542
Gwd	992	293		147.vortex	60630	1715
Gwdw	990	324		132.ijpeg	53629	6530
Tget	384	57		129.compress	2842	227
Average	3070	335		Average	35700	3390

Table 3. Dynamical Properties of ServBench and SPECint programs

The dynamical kernel size of ServBench is an order of magnitude smaller than SPECint as well, as shown in Table 3. A common rule, "90/10 rule", can be seen from the table in average: 90% of executed instructions are derived from 10% of the instructions in the program. Most of the programs have a relatively small kernel that account for most of the execution time.

4.3 Instruction Set Characteristics

The instruction mix gives indications on the types of instructions executed in the benchmark. Figure 3 presents the frequencies of different types of instructions for each ServBench program (instruction types are noted in the figure). Averages for each of the three groups of benchmarks, ServBench, and SPECint are also given in Figure 4.

These two benchmarks have similar instruction set characteristics in terms of the general trend and variability. The average difference in frequencies between ServBench and SPECint is between 5% (13% store instructions in ServBench while 9% in SPECint) and 1% (42% integer computation instructions in ServBench versus 43% in SPECint).

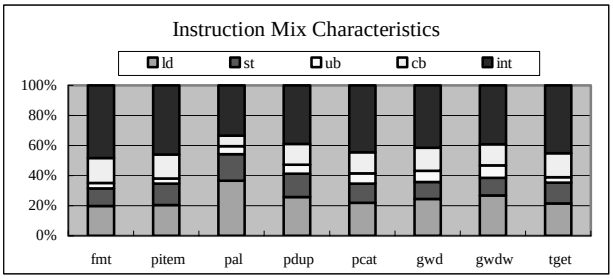


Figure 3. Instruction mix characteristics (ld=load, st=store, ub=unconditional branch, cb=conditional branch, int=integer computation. No floating-point instructions)

There are significant differences between the three groups of benchmarks, Apache, and SPECint that can be seen in from Figure 4.

The three groups of sub-benchmarks have different instruction execution frequencies. For instance, G1 has 8% percent less load instruction than G2 and 4% less than G3, however, it has 8% percent more integer computation instructions than G2 and 5% more than G3. But all these three groups have very similar percentage of store instructions. Among the three groups, string comparison functions have similar trend and variance to SPECint. The difference is under 3% (12% store instructions in group 3 compared to 9% in SPECint). Other groups have much significant and variable differences ranging from 1% to 6% in each instruction type.

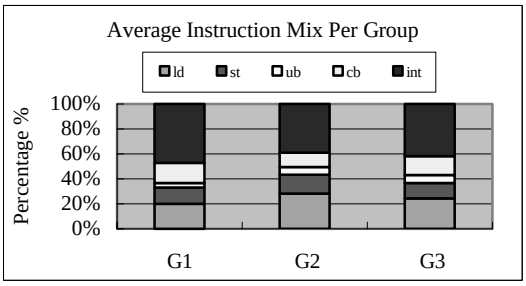


Figure 4. Average instruction mix (G1=string format conversion, G2=string generation, G3=string comparison)

4.4 Instruction Set Characteristics

Instruction level parallelism (ILP) is an important issue in improving a Web server's micro performance. Knowing the benchmarks' instruction level parallelism can be of great help to the design of application specific processors and architectures such as network processors.

We obtain the instruction level parallelism for each benchmark program by changing such parameters as the length of instruction fetch queue, the number of ALUs, and branch prediction mechanism. All these parameters have important impact.

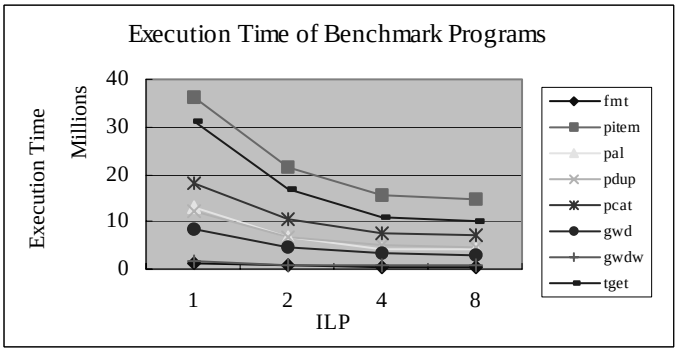


Figure 5. Execution time of benchmark programs

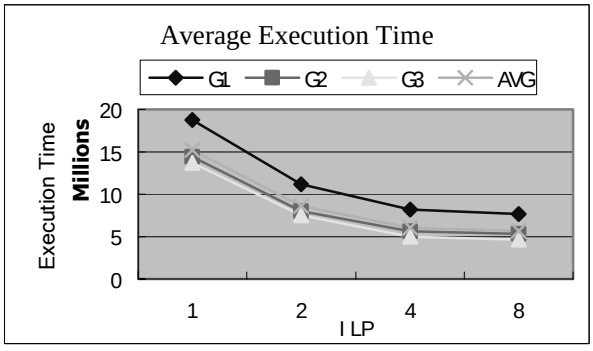


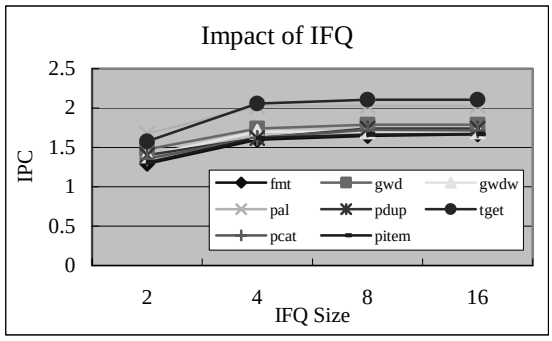
Figure 6. Avg. execution time of benchmark programs

Figure 5 shows the relationship between execution time and ILP for each benchmark program. Figure 6 depicts the average execution time of each group of functions. It is observed that it is almost enough for us to achieve the best performance when ILP is 4.

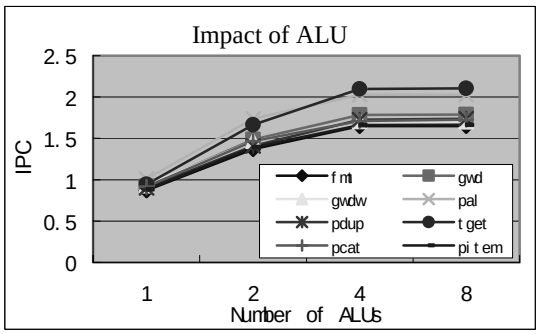
Figure 7.1 and 7.2 depict the impact of instruction fetch queue and ALU respectively.

With 8 ALUs and decode/issue bandwidth of 8 instructions per cycle, the highest instruction per cycle (IPC), which is 2.1, can be reached when instruction fetch queue is 8. Increasing the length of the queue is of no help to enhance the performance. An instruction fetch queue of length 8 is enough to achieve best performance in this case. On the other hand, 4 ALUs are enough for achieving best performance if the instruction fetch queue is 16 and decode/issue bandwidth is 8. Increasing the number of ALU does not help to improve the performance in terms of IPC.

From the above observations, we see that 4 ALUs and instruction fetch queue of length 8, or an ILP of 4, are enough for best performance. However, there are intrinsic reasons for this.



(1) Impact of instruction fetch queue



(2) Impact of ALU

Figure 7. Impact of instruction fetch queue and ALU

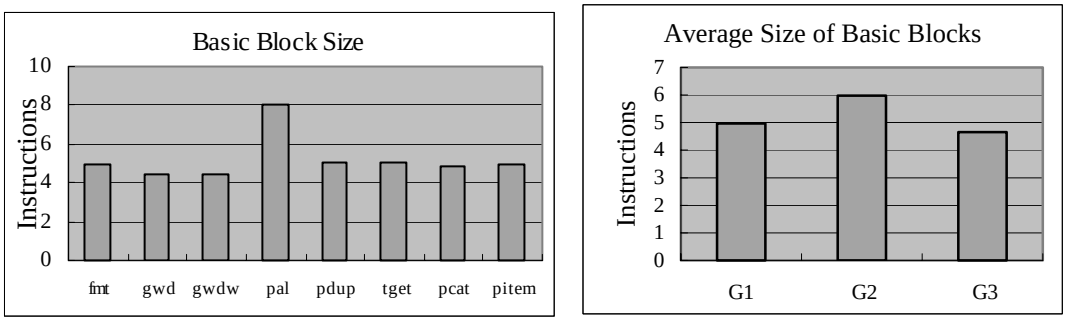
Figure 8 shows the size of basic blocks of each benchmark program and group. Most of the programs' basic block sizes are less than 5 except pal which has the size of 8. The three groups of programs have average size of 5, 6, and 4.6 respectively. This means every 5 or 6 instructions in the instruction queue must have a branch which is taken with a probability of nearly 50%, as shown in Figure 9. Thus an instruction queue of length 8 has an effective length of 4 due to half of the branches are taken and the other half not taken. 4 ALUs are enough for best performance for two reasons. One reason is that the effective length of the instruction queue is only 4 which means there are at most 4 instructions decoded and issued to the ALUs. The other reason is that only 40% instructions are integer computation instructions and that the maximum decode/issue bandwidth is 8 instructions per cycle, which means less than 4 instructions per cycle are in need of ALU operations.

The miss rate of branch prediction mechanism has much greater impact on IPC than expected. Both predict-not-taken and predict-taken have nearly the same high miss rate as shown in Figure 9.1. The bimod, 2lev and combined techniques predict with an accuracy between 80% to 100% for different benchmarks. From Figure 9.2 we can see that IPC reaches more than 2 for most of the programs with perfect branch prediction mechanism. However, even with the best branch-prediction mechanism, combining bimod with 2lev, IPC can only reach 86% of IPC with perfect branch prediction. There is much room for improving micro performance by means of improving branch prediction hit rate.

4.5 Instruction Set Characteristics

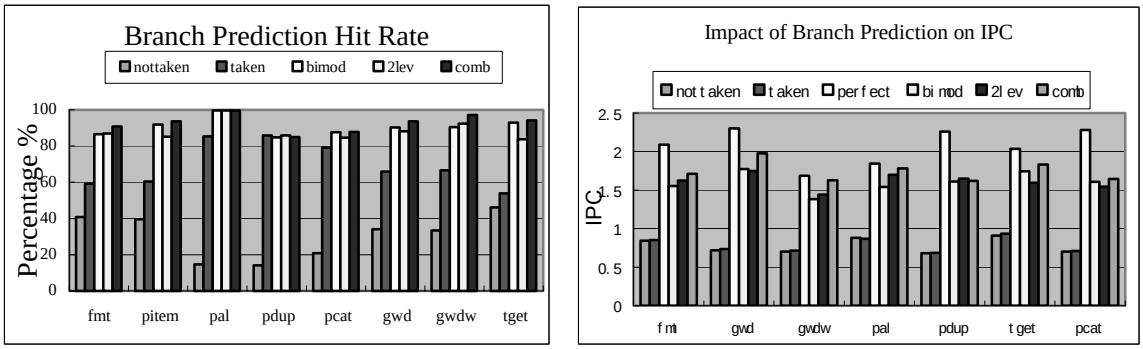
Cache behavior is very important to the micro performance. We measured the cache performance for each ServBench program. Separate L1 instruction cache and data cache were simulated. The size of data cache ranges from 2KB to 256KB, that of instruction cache ranges from 2KB to 64KB in Figure 10 and Figure 11, which show the miss rates for a 4-way associative data cache and instruction cache respectively. The results are shown in terms of groups.

It seems that the size of instruction cache has greater impact than data cache. The instruction miss rates for small caches sizes are much higher than the corresponding data cache miss rates. When cache size increases from 16KB to 32KB, data cache miss rate decreases 43% in average (26% for G1, 34% for G2, and 58% for G3), however, instruction cache miss rate decreases 76% in average (52% for G1, 25% for G2, and 98% for G3). When cache size increases from 32KB to 64KB, miss rates of data cache and instruction cache decrease 23% and 63% respectively in average. Large instruction cache favors ServBench.



(1) Basic-block size per program (2) Basic-block size per group

Figure 8. Size of Basic Block of ServBench programs



(1) Branch-prediction Hit Rate (2) Impact of Branch Prediction on IPC

Figure 9. Impact of Branch Prediction

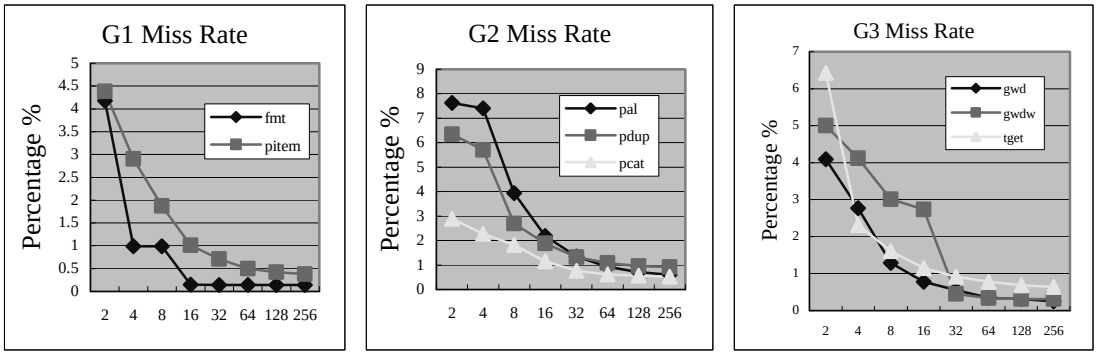


Figure 10. L1 Data Cache Miss Rate as a function of cache size

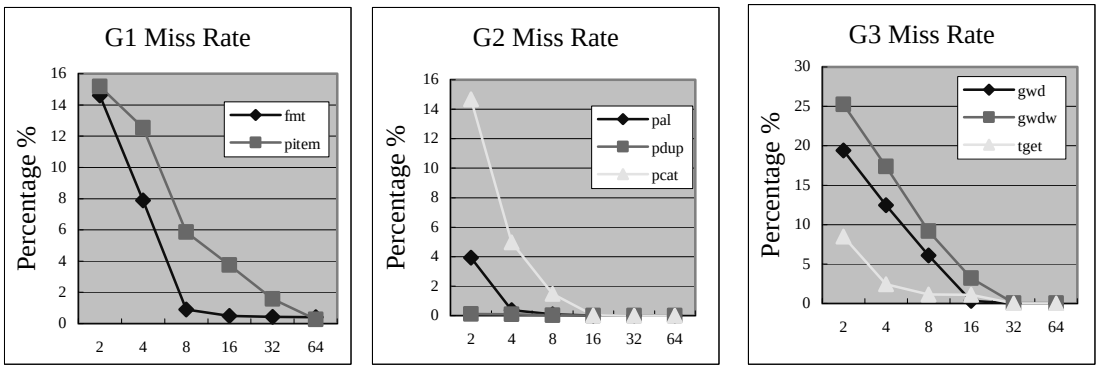


Figure 11. L1 Instruction Cache Miss Rate as a function of cache size

Figure 12 compares the ServBench, SPECint, and the three groups of benchmark programs. We need at least 16KB instruction cache to obtain the miss rate under 2% and 32KB to lower the miss rate to 1%. Although the kernels of ServBench programs are small, there are a lot of standard library functions called by the kernels which makes the instruction cache miss rate higher than expected. Compared to SPECint, only G2 programs, which have the smallest kernel, have lower miss rate.

Compared to instruction cache behavior, data cache performance of ServBench is more similar to that of SPECint. The data cache miss rates for ServBench are roughly half that of SPECint.

4.6 Instruction Set Characteristics

It seems that L1 instruction cache has greater impact on the micro performance. From cache performance results we observe larger instruction cache favors ServBench. We obtain memory access behavior measured by memory accesses per instruction (MAPI), as shown in Figure 13.

Figure 13 shows the memory access behaviors of benchmarks and Apache. We define memory access per instruction (MAPI) as the ratio of number of memory references for data to the number instructions executed in a run. MAPI represents the frequency of memory accesses in terms of instruction execution. ServBench has a similar MAPI as SPECint with a variance of less than 3%.

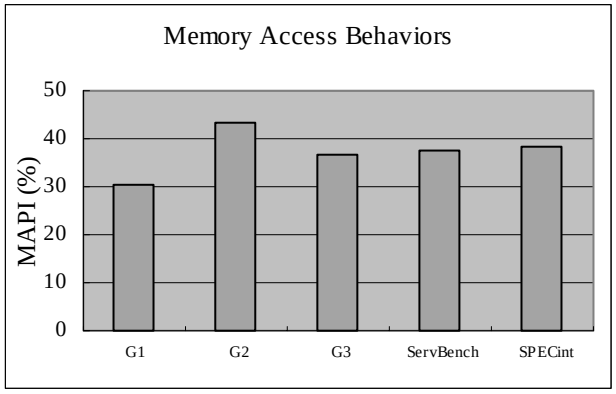


Figure 13. Memory Access Behaviors

Based on the above results and observations, it is possible to use asymmetric L1 caches to improve the micro performance. Most of present processors have symmetric L1 caches. For example, the mainstream microprocessor, Pentium II processor, has symmetric L1 instruction cache and data cache, both of which have 16KB. However, asymmetric L1 caches are more suitable to improve the performance. Figure 14 depicts the different impacts of L1 instruction cache and data cache on IPC. Increasing the size of L1 instruction cache contributes 60% performance improvement when the size ranges from 8KB to 128KB.

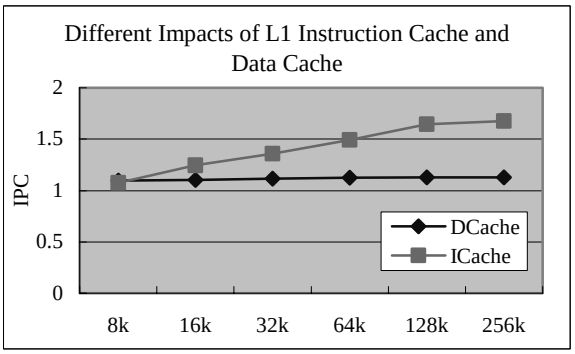


Figure 14. Different Impacts of L1 Caches on IPC

Based on the above observations, when we combine the results of Table 2 and Table 3 with those in Figure 2, we propose that these micro kernels be put in a part of the instruction cache which is not replaced to make room for other instructions.

5. Conclusions

This paper has presented a micro-benchmark, ServBench, for use in benchmarking the micro performance of Web servers. All the benchmark programs are taken from the implementation of the most commonly used Apache Web server by measuring and profiling the server with realistic workloads. We do not consider network and disk I/O functions for the reason that there has already been extensive research in decreasing and optimizing the network and I/O latency. All the dataset for the benchmark are obtained from the realistic workloads. Then we port the micro-benchmark and Apache Web server to SimpleScalar simulation environment. We obtain execution time, branch prediction and cache miss results for the micro-benchmark as a function of various architecture parameters.

The average code size of ServBench is an order of magnitude smaller than SPECint. Both have similar instruction set characteristics. ServBench has smaller basic-block sizes and nearly half of the branches are taken and half not taken. This fact makes better branch predication mechanism and lower miss rate very important to the performance.

By comparing ServBench and SPECint, we observe that L1 instruction cache plays a more important role than data cache in improving micro performance. We prove this observation by porting and running Apache in SimpleScalar simulation environment. We find that instructions executed per cycle are increased by 60% when the size of L1 instruction cache increases from 8KB to 128KB. Asymmetric L1 caches help to improve micro performance greatly. We are able to achieve higher IPC by using asymmetric L1 cache and enlarging the length of

instruction fetch queue and adding more ALUs. However, 4 ALUs and an instruction fetch queue of length 8 are enough to enhance the micro performance.

References

[1] M. Almeida, V. Almeida, D.J. Yates, Measuring the Behavior of A World-Wide-Web Server, 7th IFIP Conference on High Performance networking (HPN), White Plains, NY, Apr. 1997

[2] Apache, http://www.apache.org/

[3] M.F Arlitt, C.L. Williamson. Web Server Workload Chracterization: The Search for Invariants, Proceeding of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, pages 126-137, 1996

[4] G. Banga, P. Druschel, Measuring the Capacity of a Web Server, Proceedings of the USENIX Syposium on Internet Technologies and Systems, Monterey, Dec 1997

[5] G. Banga, P. Druschel, J. C. Mogul. Better Operating System Features for Faster Network Servers, Proceedings of the Workshop on Internet Server Performance, Madison, WI, June 1998

[6] G. Banga, J.C. Mogul, Scalable Kernel Performance for Internet Servers Under Realistics Loads, Proceedings of 1998 Usenix Annual Technical Conference, New Orleeans, LA, June 1998

[7] P. Barford, M. Crovella, Generating Representative Web Workloads for Network and Server Performance Evaluation, Proceeding of the ACM SIGMETRICS'98 Conference, Madison, WI, 1998

[8] D. Burger, T.M. Austin, The SimpleScalar Tool Set, Version 2.0, Technical Report, Computer Science Department, University of Wisconsin-Madison, June 1997

[9] S. Glassman, A Caching Relay for the World Wide Web. WWW'94 Conference Proceedings, 1994

[10] N. Gloy, C. Young, J. Chen, M. smith, An Analysis of Dynamic Branch Prediction Schemes on System Workloads, Proceedings of the International Symposium on Computer Architecture, May 1996

[11] S.L. Graham, P.B. Kessler, M.K. McKusick, gprof: A Call Graph Execution Profiler, Proceedings of the SIGPALN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982

[12] GNU Unix Toolset. Information and binaries available at http://www.gnu.org/

[13] V. Holmedahl, B. Smith, and T. Yang, Cooperative caching of dynamic content on a distributed web server, Proceedings of 7th IEEE International Symposium on High Performance Distributed Computing (HPDC-7), Chicago, IL USA July 28-31, 1998.

[14] Y. Hu, A. Nanda, Q. Yang, Measurement, Analysis and Performance Improvement of the Apache Web Server, the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99), Phoenix/Scottsdale, Arizona, February 1999

[15] C. Huitema, Network vs. Server Issues in End-to-end Performance, Keynote Speech, Performance and Architecture on Web Servers (PAWS), June 2000

[16] R. Iyer, Exploring the Cache Design Space for Web Servers, Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'00), San Francisco, CA , April 2000

[17] S. Manley, M. Seltzer, M. Courage. A Self-Scaling and Self-Configuring Benchmark for Web Servers. Proceeding of the ACM SIGMETRICS'98 Conference, Madison, WI, 1998

[18] D. Mosberger, T. Jin, httperf---A Tool for Measuring Web Server Performance, Workshop on Internet Server Performance (WISP98), Madison, Wisconsin, June 23, 1998

[19] E. Nahum, T. Barailai, D. Kandlur, Performance Issues in WWW Servers, Proceedings of the international conference on Measurement and modeling of computer systems, 1999

[20] NetPerf, http://www.netperf.org/

[21] V. Pai, P. Druschel, W. Zwaenepoel, Flash: An Efficient and Portable Web Server, Proceedings of the USENIX 1999 Annual Technical Conference, Monterey, CA, June 1999

[22] V. Pai, P. Druschel, W. Zwaenepoel, IO-Lite: A Unified I/O Buffering and Caching System, ACM Transactions on Computer Systems, Vol. 18, No. 1, pp.37-66, February 2000

[23] R. Radhakrishnan, L.K. John, A Performance Study of Modern Web Applications, Euro-Par 1999, Lecture Notes in Computer Science, Springer, pages. 239-247, 1999

[24] SPECWeb99 Benchmark, http://www.spec.org/osg/web99

[25] Standard Performance Evaluation Corporation, SPEC CPU95 Version 1.10, August 21, 1995

[26] G. Trent, M. Sake, WebStone: the First Generation in HTTP Server Benchmarking, White Paper, Silicon Graphics, Feb 1995

[27] WebBench, http://www.webbench.com/, Ziff Davis, Inc. March 2000

[28] D.J. Yates, V. Almeida, J.M. Almeida, On the Interaction Between an OS and Web Server, Boston University Computer Science Department, Boston Univ., MA, Tech Report CS 97-012, July 1997